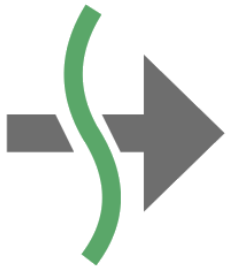
CMPS 312

Coroutines for Asynchronous Programming



Dr. Abdelkarim Erradi

CSE@QU

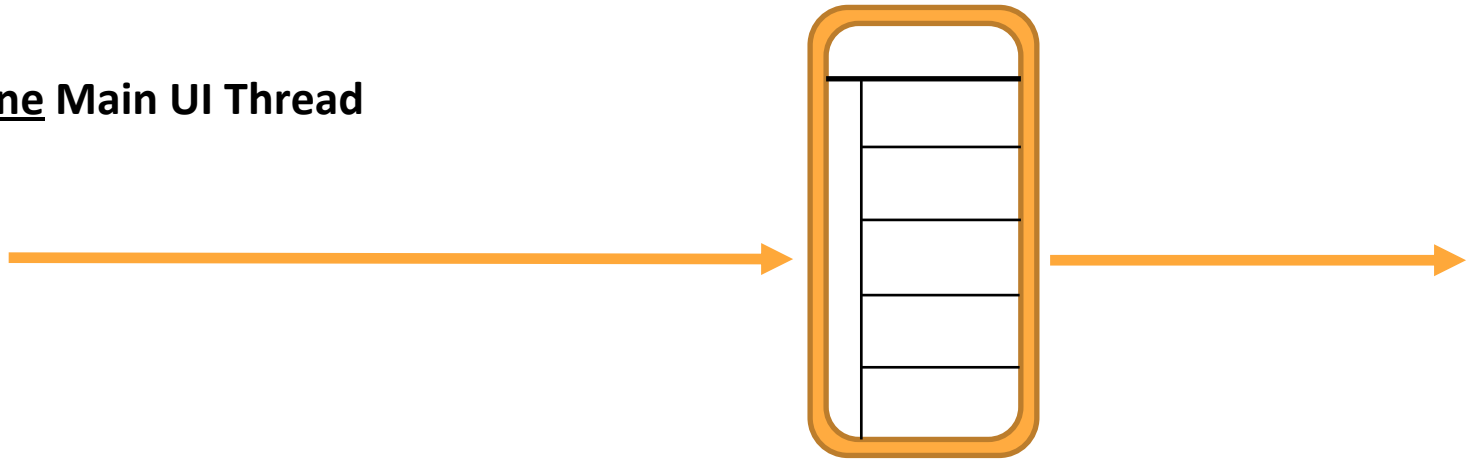
Outline

1. Coroutines Basics
2. Coroutines Programming Model
3. Coroutine Cancelling
4. Exception Handling

Coroutines Basics

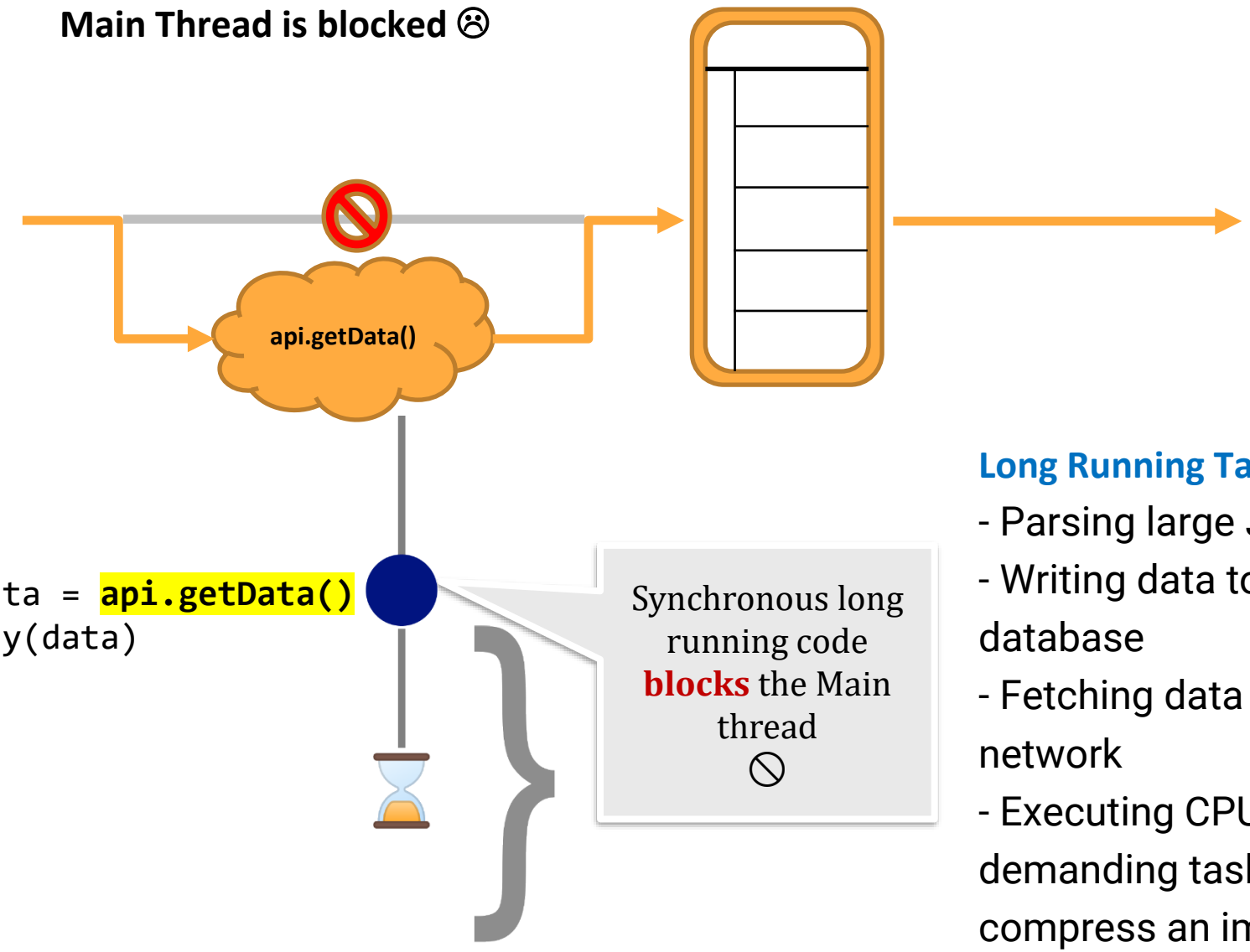
User Interface Running on the Main Thread

One Main UI Thread



To guarantee a great user experience, it's essential to **avoid blocking the main thread** as it used to handle UI updates and UI events

Long Running Task on the Main Thread



Long Running Tasks include:

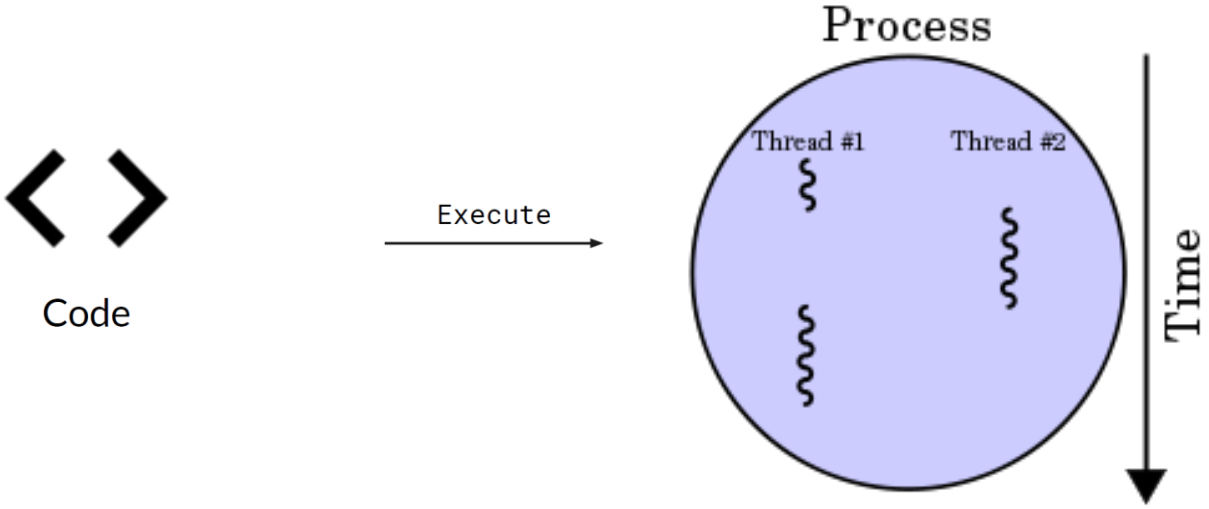
- Parsing large JSON file
- Writing data to a database
- Fetching data from the network
- Executing CPU demanding task (e.g., compress an image)

How to address problem of long-running task?

- How to execute a long running tasks without blocking the Main thread?

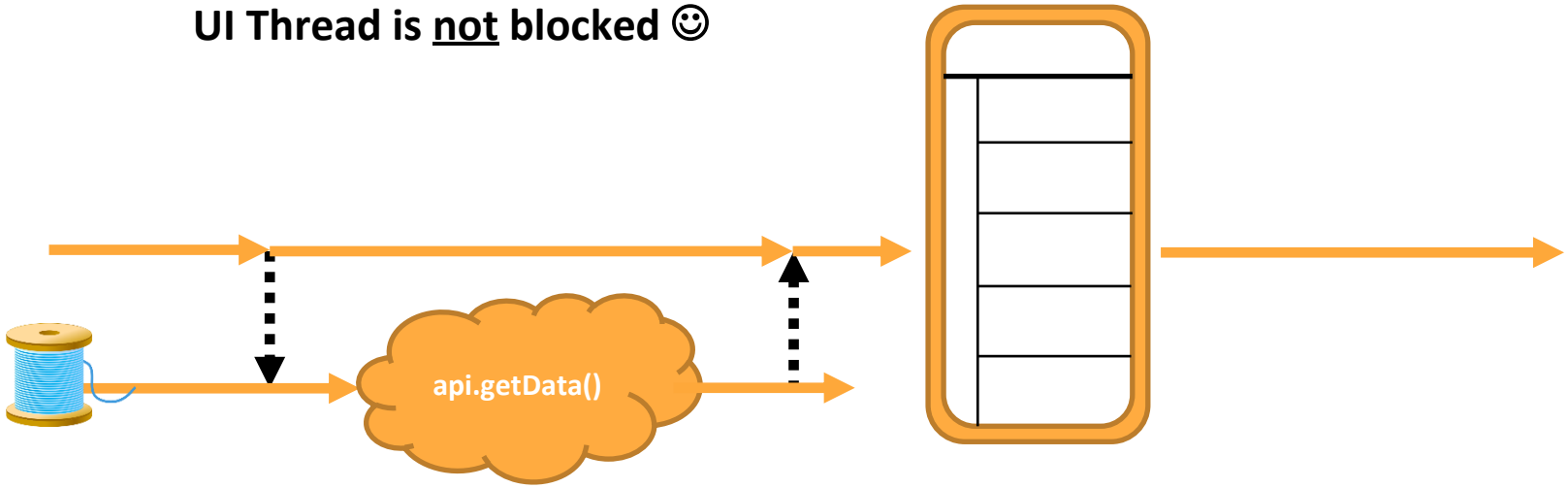
  => Solution 1: **Use multi-threading**

- A thread is the **unit of execution** within a process
  - It allows **concurrent** execution of tasks within an App



Solution 1 – Run Long Running tasks on a background thread

UI Thread is <u>not</u> blocked ☺




```
thread {
    val = result = api.getData()
}
```

- UI can only be accessed from the Main thread
- How to transfer the result from the background thread to the main thread?

How to transfer the result from the background thread to the main thread?

- By using callbacks, you can start long-running tasks on a background thread
- When the task completes, the callback is called to notify the main thread of the result

Limitations:

- Nested callbacks can become difficult to understand (aka **Callback Hell**)
- Difficult to cancel background tasks
- Difficult to run tasks in parallel
- Difficult to handle exceptions

Callback Example

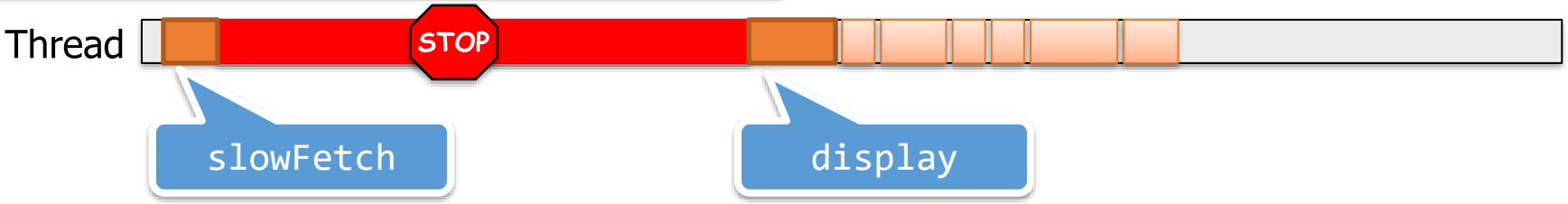
```
fun main() {
    // Call the function and pass callback function
    getUserOrders("sponge", "bob") { orders ->
        orders.forEach { println(it) }
    }
}
fun getUserOrders(username: String, password: String,
                  callback: (List<Order>) -> Unit) {
    login(username, password) { user ->
        fetchOrders(user.userId) { orders ->
            // When the result is ready, pass it to main using the callback
            callback(orders)
        }
    }
}
```

Callback hell = Nested callback functions are difficult to understand

Synchronous vs. Asynchronous Functions

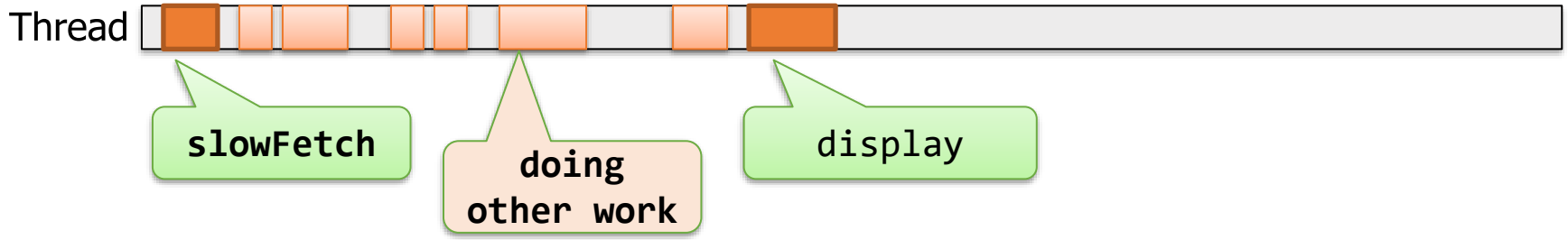
```
val result = slowFetch(...) // UI Thread
display(result)  // UI Thread
```

Synchronous (i.e. **Blocking**) → Wait for result before returning




```
// Slow request with callbacks
fun makeNetworkRequest(display: (Result) -> Unit) {
 // The slow network request runs on another thread
 thread {
    slowFetch { result ->
       // When the result is ready, this callback will get the result
       display(result)
    }
  }
}
```

Asynchronous (i.e. **Non-Blocking**) → do an **asynchronous** call to slowFetch using backgroud thread, then update UI with the result



UI not blocked

Thread Limitations

Threads are **costly** (occupy 1-2 mb)

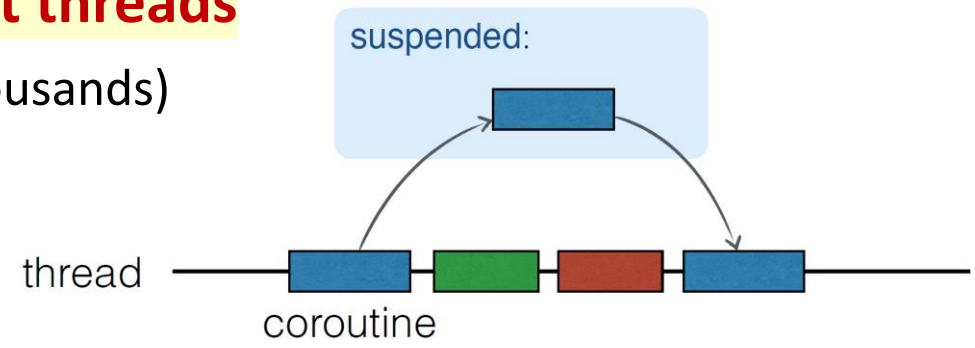
- Some **threads** are **special** (e.g. Main UI thread) and should not be blocked

Better alternative are **Coroutines**

- Coroutines are like light-weight threads

(very cheap and fast to create even thousands)

- Coroutine = computation that can be suspended then resumed



Thread is not blocked!

Why Coroutines?

Most mobile apps typically need:

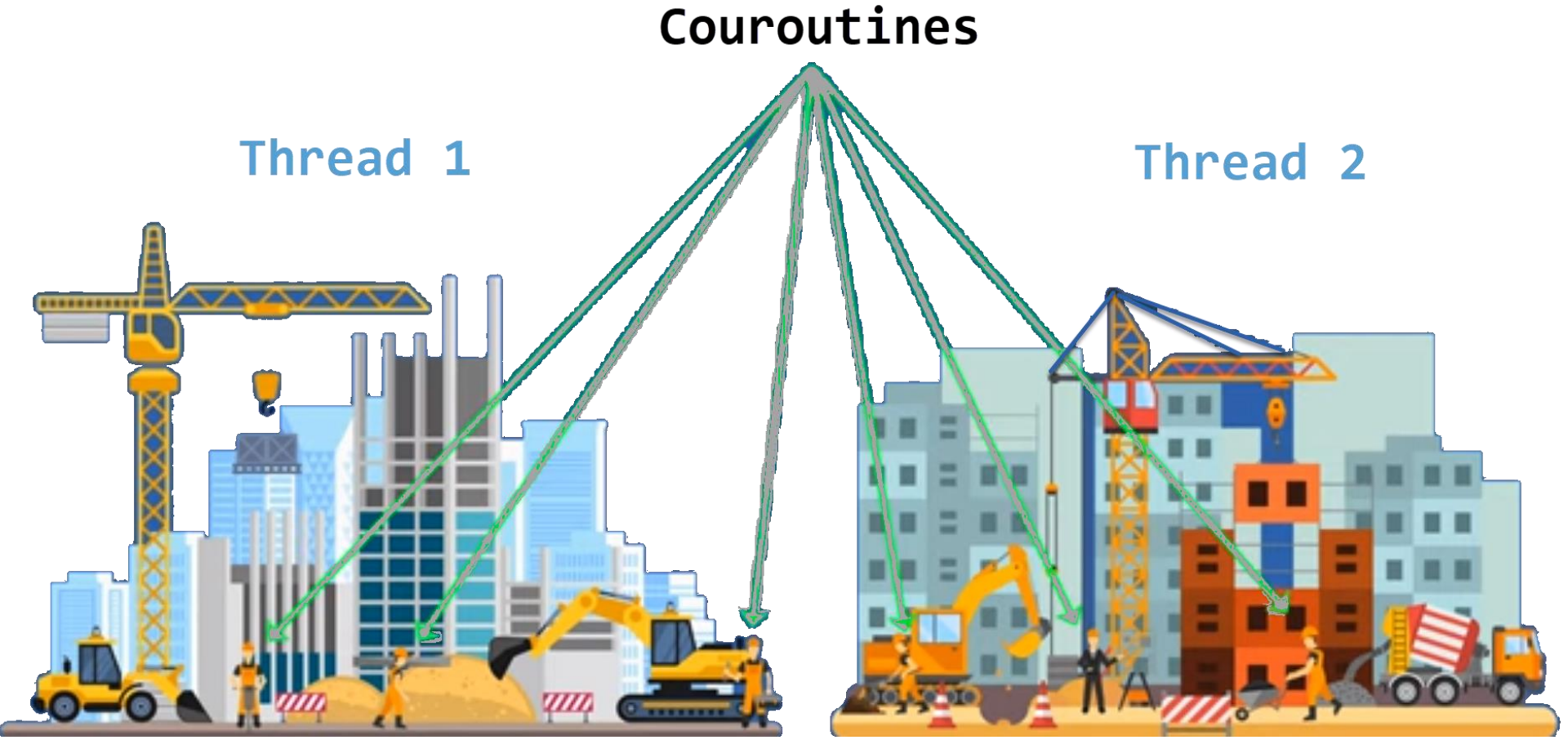
Call Web API (Network Calls)	Database Operations (read/write to DB)	Complex Calculations (e.g., image processing)

Can use coroutines to offload long-running computations or Asynchronous I/O operations without blocking the main UI thread

What distinguishes Coroutines from Threads?

1. Coroutines are like **light-weight** threads. They are more efficient and yield better performance
   - Multiple coroutines can run within a thread
2. Easier **cancellation** of a long running coroutine
3. Easier **exception handling**
4. Easier to **run coroutines in parallel** to improve the app performance
5. Easier to **switch the coroutine execution between threads**
   - e.g., do a Network call using the IO Thread then switch to the Main thread to update the UI
6. Easier **asynchronous** programming
   - Replace callback-based code with sequential code to handle asynchronous long-running tasks without blocking

Thread vs. Coroutine



Multiple (even thousands) coroutines can run within a thread

Analogy: multiple workers at a construction site

Source: https://www.youtube.com/watch?v=ShNhJ3wMpvQ

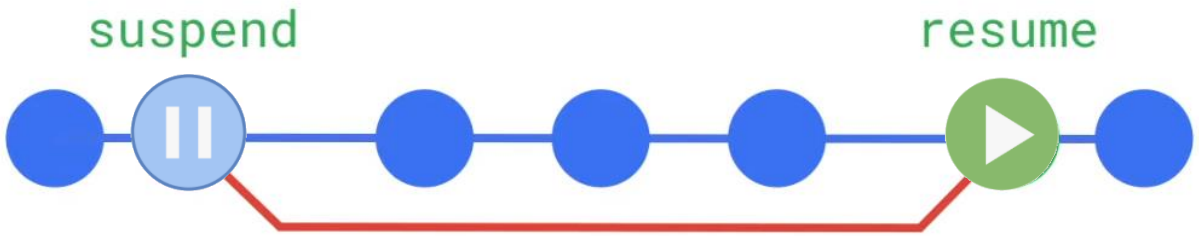
Callback vs. Coroutine

- Compared to callback-based code, coroutine code accomplishes the same result of unblocking the main thread **with less code**.
- Due to its sequential style, it's **easier to understand** + it's easy to chain several long running tasks without creating multiple callbacks

```
fun getUserOrders(username: String, password: String, callback: (List<Order>) -> Unit) {
    login(username, password) { user ->
        fetchOrders(user.userId) { orders ->
            // When the result is ready, pass it to main() using the callback
            callback(orders)
        }
    }
}
```

```
suspend fun getUserOrders(username: String, password: String) =
    withContext(Dispatchers.IO) {
        val user = login(username, password)
        val orders = fetchOrders(user.userId)
        orders
    }
```

Coroutines Programming Model



Blocking vs. Non-Blocking (suspendable task)

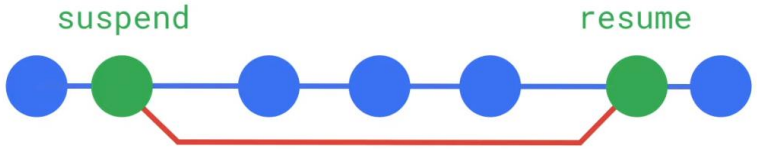
- **Salat**  **vs.**  **Reading a Book** 

Mum: 📣 “Fatima comedown dinner ready!”

=> Salat is a **bocking** task. The caller needs to wait for Salat to complete to get an answer

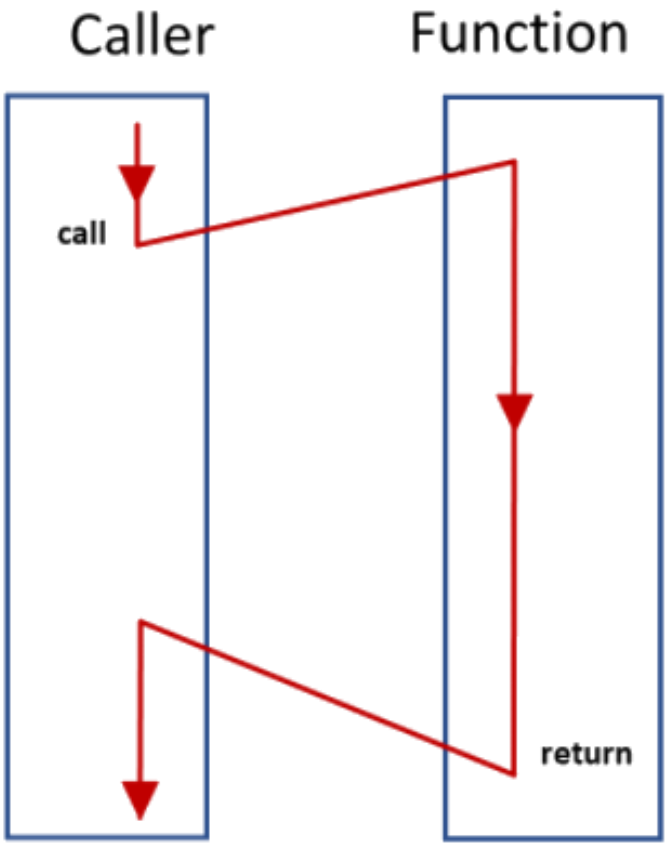
=> Reading a book is a non-blocking task than can be **suspended** then **resumed**: add a bookmark then suspend reading, when ready resume reading from the bookmark

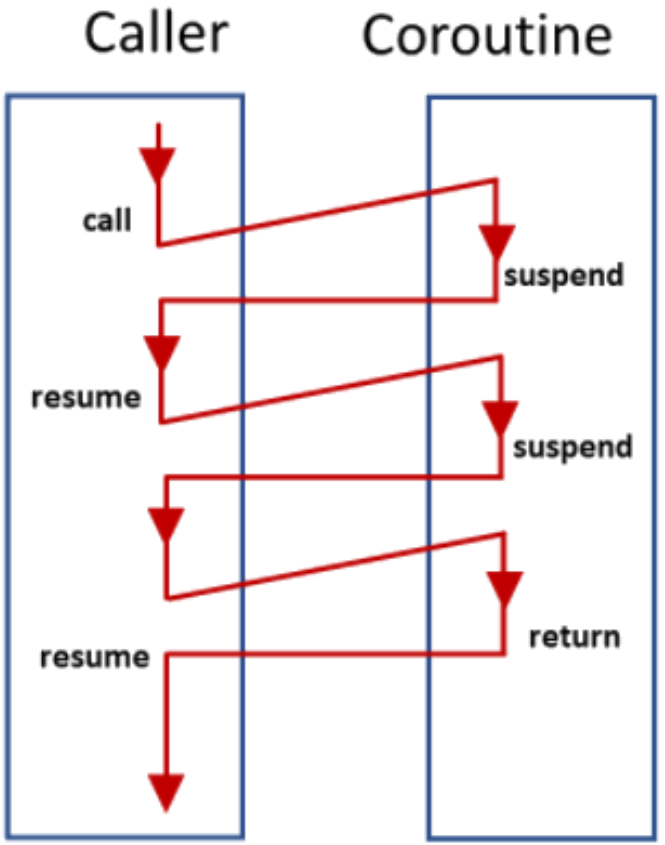
Suspend function



- **Suspend** function is a function that can be **suspended** and **resumed**
  - **suspend** functions are used to create coroutines
- When a suspend function needs to wait for a result it does NOT block instead the runtime:
  - **suspends** the function execution, removes it from the thread, and stores the state and the remaining function statements in memory until the result is ready then
  - **resumes** the function execution where it left off
- While it's suspended waiting for a result, **it unblocks the thread that it's running on**, so that the thread is free to be used for other tasks

Function vs. Suspend Function



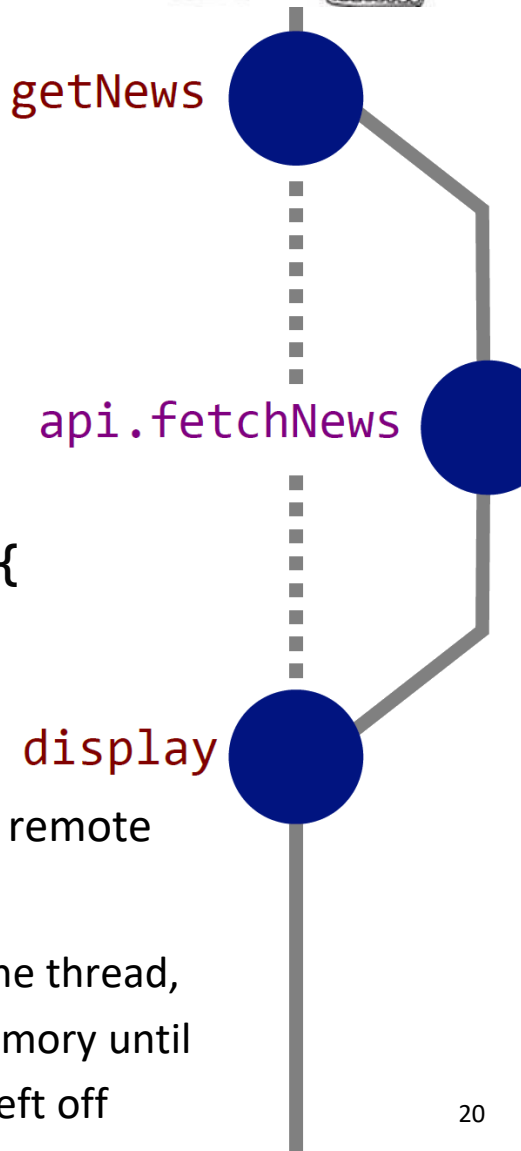


Suspend function can **suspend** at some points and later **resume** execution (possibly on another thread) when the return value is ready

Async Non-blocking calls with Coroutines

```
val coroutineScope = rememberCoroutineScope()
Button(
    onClick = {
        coroutineScope.launch {
            newsStateVar = getNews()
        }
    }) {
    Text(text = "Get News")
}

suspend fun getNews() = withContext(Dispatchers.IO) {
    return api.fetchNews()  // IO thread
}
```



- When getNews suspend function is waiting for the result from the remote news service it does NOT block instead the runtime:
  - suspends the execution of getNews() function, removes it from the thread, and stores the state and the remaining function statements in memory until the result is ready then resumes the function execution where it left off

To launch a Coroutine you need a Coroutine Scope

- A suspend function must be called in a coroutine
- A **Coroutine Scope** is required to create and start a coroutine using the scope's **launch** or **async** methods
- Coroutine Scope keeps track of child coroutines to allow the ability to cancel them and to handle exceptions
- Can be created as an instance of ***CoroutineScope***

```
val coroutineScope = CoroutineScope(Dispatchers.IO)
coroutineScope.launch { }
```

- On Android you could use provided scoped:
  - ***viewModelScope***, ***lifecycleScope***, ***rememberCoroutineScope***
  - GlobalScope is an app-level scope (rarely used). It lives as long as the app does

viewModelScope

- **viewModelScope** can be used in any ViewModel in the app
- Any incomplete coroutine launched in this scope is **automatically canceled** if the ViewModel is cleared (to avoid consuming resources unnecessarily)

```
class MyViewModel: ViewModel() {
    init {
        viewModelScope.launch {
            // Incomplete Coroutine will be canceled when the ViewModel is cleared
        }
    }
}
```

lifecycleScope

- **lifecycleScope** can be used in an activity
- Any incomplete coroutine launched in this scope is canceled when the Lifecycle is destroyed

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        lifecycleScope.launch {
         // Incomplete coroutines will be canceled when the activity is destroyed
        }
    }
}
```

LaunchedEffect

- `LaunchedEffect` should be used to execute some action when the **composable** is first launched
  - For example, requesting some data from the ViewModel
- With **LaunchedEffect**, you cannot control the lifecycle of the coroutine
  - The coroutine starts and ends based on the Composable lifecycle and has no way to manually cancel it

rememberCoroutineScope

- Use `rememberCoroutineScope` to create a **CoroutineScope** bound to the Composable lifecycle
  - If the composable leaves the recomposition, the coroutine will be cancelled automatically

```
/* Create a CoroutineScope that follows
   this composable's lifecycle */
val coroutineScope = rememberCoroutineScope()

coroutineScope.launch {
    //... your code
}
```

Coroutine builder functions

A coroutine scope offers two builder functions to **create** and **start** a coroutine

- **`launch`** **`Fire and forget`**

```
scope.launch(Dispatchers.IO) {
    loggingService.upload(logs)
}
```

- **`async`** **`Returns a value`**

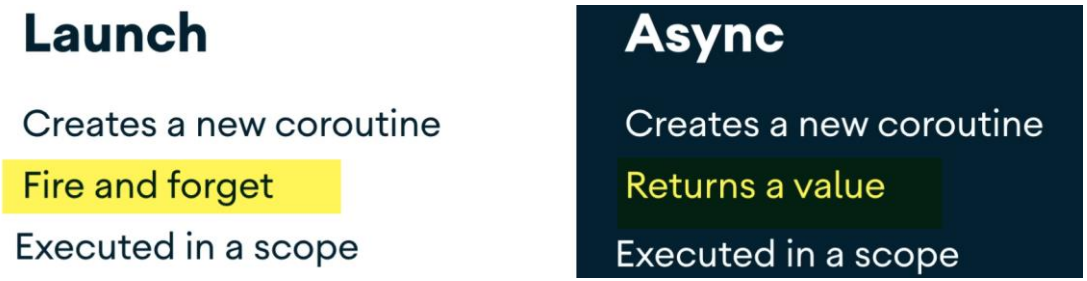
```
suspend fun getUser(userId: String): User =
    coroutineScope {
        val deferred = async(Dispatchers.IO) {
            userService.getUser(userId)
        }
        deferred.await()
    }
```

Launch vs. Async Coroutine builder functions

- **Launch** - Launches a new coroutine and returns a **job** which can then be used to **cancel** the coroutine
- **Async** - Launches a new coroutine and returns its future result (of type **Deferred**)

```
val deferred = async { viewModel.getStockQuote(company) }
```

  - Can use `deferred.await()` to suspend until the result is ready
  - Or call `deferred.cancel()` to cancel the coroutine



Parallel Execution of Coroutines

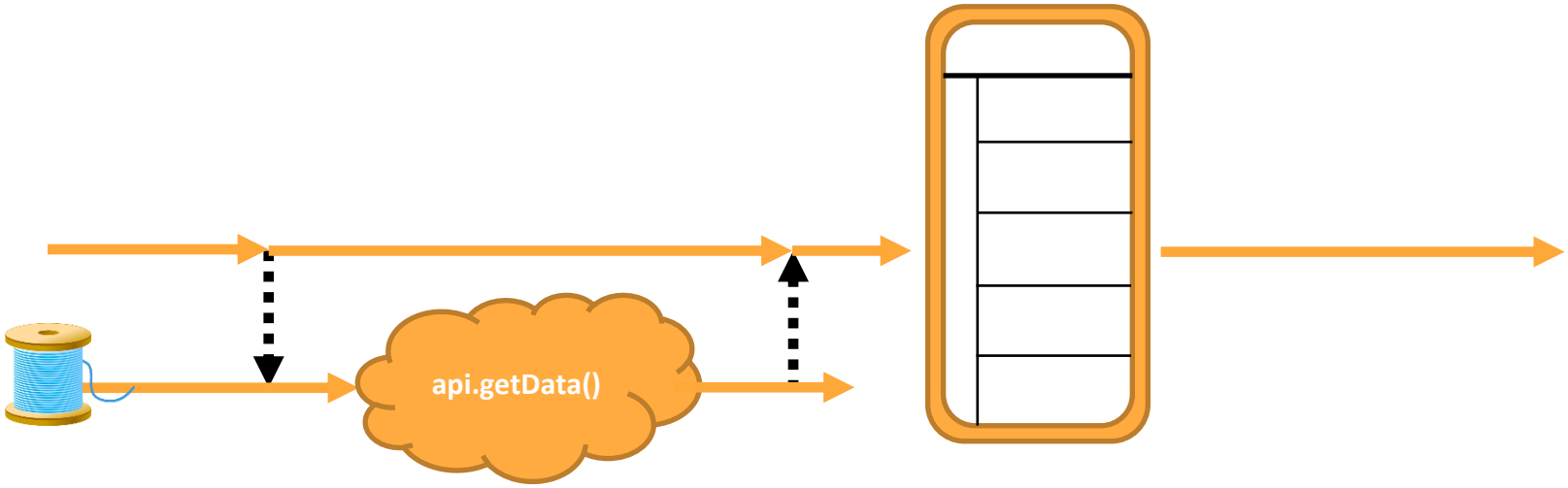
- Coroutines can be **executed in parallel** (concurrently) using **Async** or **Launch**
  - Parallelism is about doing lots of things simultaneously
- Async can await for the results (i.e. suspend until results are ready)

```
val deferred1 = async { getStockQuote("Apple") }
val deferred2 = async { getStockQuote("Google") }

val quote1 = deferred.await()
println(">> ${quote1.name} (${quote1.symbol}) = ${quote1.price}")

val quote2 = deferred2.await()
println(">> ${quote2.name} (${quote2.symbol}) = ${quote2.price}")
```

Switch between threads



Perform fetch data on background thread then when the result is ready update the UI on Main thread

```
coroutineScope.launch(Dispatchers.Default) {
    val value = fibonacci(1000)
    withContext(Dispatchers.Main) {
        resultStateVar = value.toString()
    }
}
```

Switch to Main Thread to update the UI state variable

Switch between threads

```
coroutineScope.launch(Dispatchers.?) {

withContext(Dispatchers.?) {  ...  }
```

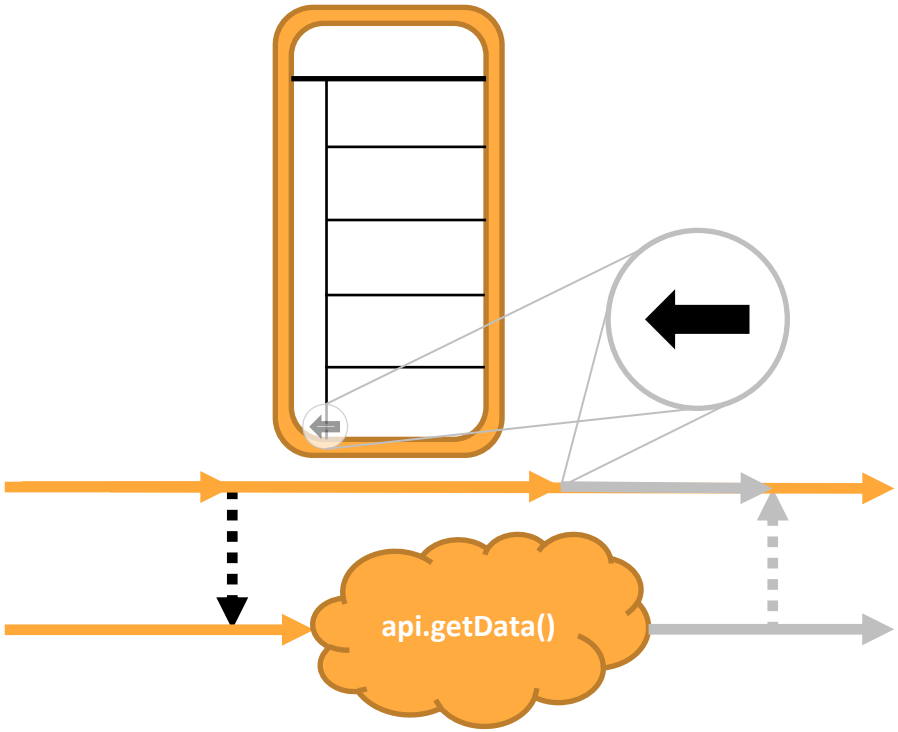
- `coroutineScope.launch` & `withContext` allows you to *decide the **thread** where* to run the computation
  - **`Dispatchers.IO`**: Optimized for Network and Disk operations
  - **`Dispatchers.Default`**: used form CPU-intensive tasks
  - **`Dispatchers.Main`**: Used for updating the UI
- Use **`withContext`** to switch between threads

Coroutine Cancelling

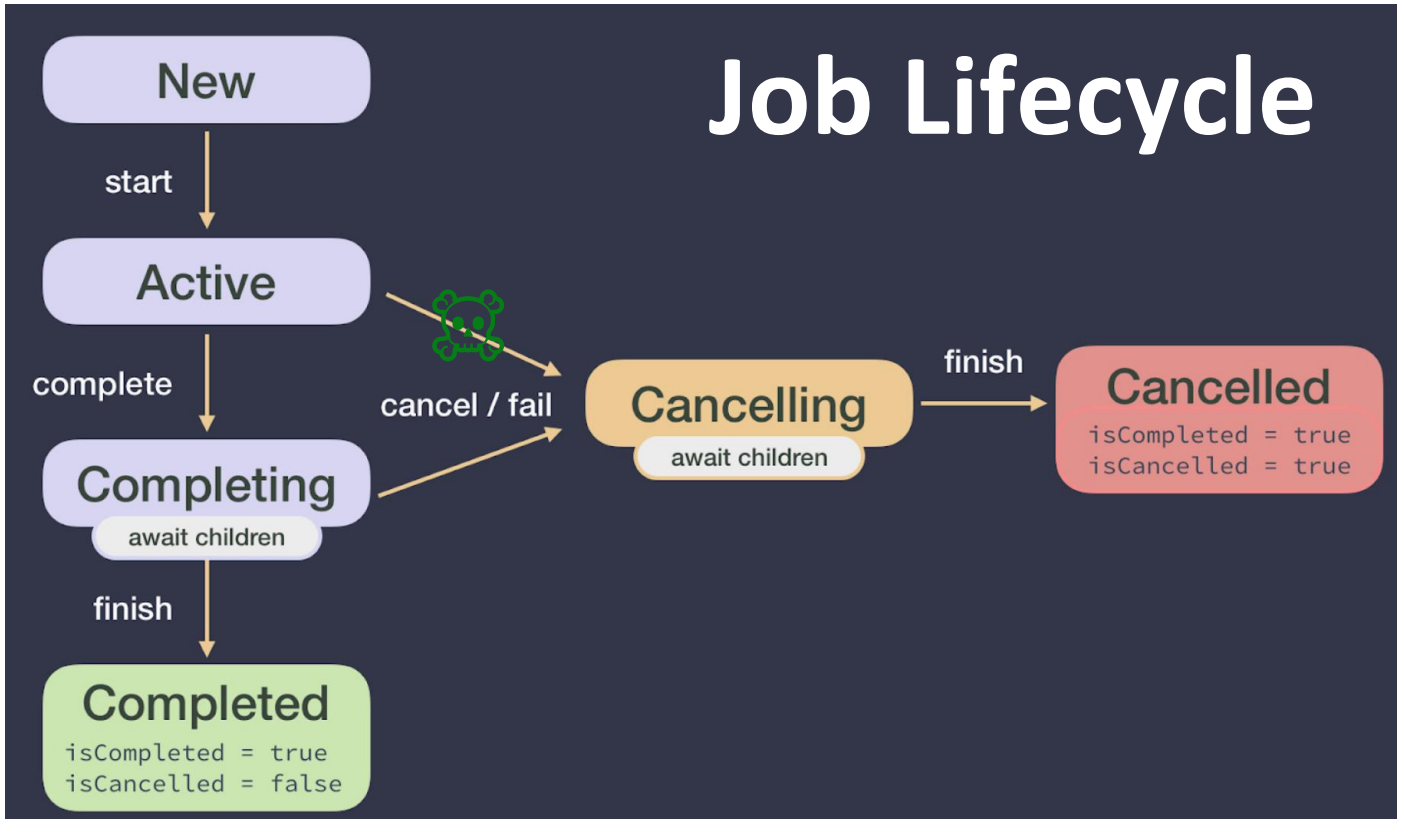
Coroutine Scope allows Structured Concurrency

- A coroutine **must** be **started in a Coroutine Scope** to allow **Structured Concurrency** i.e., the coroutine scope:
  - Keeps track of coroutines running in parallel or sequential
  - Has the ability to cancel running coroutines
  - Is notified when a failure happens: the failure of a child coroutine cancels the scope & other children
- Coroutines started in the same scope form a **hierarchy** (scope is the parent and coroutines are children) having these properties:
  - A parent job **won't complete, until all its children have completed**
  - **Cancelling a parent will cancel all children**
  - Cancelling a child won't cancel the parent or siblings
  - If **a child coroutine fails**, the exception is propagated upwards, and **all the incomplete siblings are cancelled** (unless if a **supervisorScope** is used)

Coroutine Cancelling



- When the View is destroyed (e.g., Back Button pressed). How to cancel `api.getData()` task?
- Otherwise, **waste memory and battery life** + possible memory leak of UI that listens to the result of `getData()`




```
// Coroutine Scope enables Cancellation

val job = lifecycleScope.launch(Dispatchers.Default) {
    fibonacci()
}
...
// onCancel button clicked
job.cancel()
```

Coroutine Cancelling

```
// Create a coroutineScope and run multiple jobs
val scope = CoroutineScope(Dispatchers.IO)
val job1 = scope.launch { … }
val job2 = scope.launch { … }

// Cancelling the scope cancels its children
scope.cancel()

// Or you can cancel a particular job
// First coroutine will be cancelled and the other
// one won't be affected
job1.cancel()
```

```
val JOB_TIMEOUT = 5000L
// Cancel the job after 5 seconds timeout
// job will be null if the job is cancelled
val job = withTimeout(JOB_TIMEOUT) {
    fibonacci().collect {
        print("$it, ")
    }
}
```

Cancellation is cooperative

- A coroutine code has to cooperate to be cancellable
  - If a coroutine is working in a computation and does not check for cancellation, then it cannot be cancelled
- 3 ways to make a coroutine cancellable by checking if job was cancelled and if so, exit the coroutine using either (see `fibonacci() example`):
  - **`ensureActive()`**
  - **`yield()`**
  - **`if (!job.isActive)`** `return`

Exception Handling

Exception Handling

```
/*  By default, if one child failed the whole job is cancelled
     and all incomplete sibling jobs are cancelled.
     Unless supervisorScope is used (see example 12) */
val exceptionHandler = CoroutineExceptionHandler { context, exception ->
    println("Exception thrown somewhere within parent or child: $exception.")
}
val job = CoroutineScope(Dispatchers.IO).launch(exceptionHandler) {
    val deferred1 = async() { getStockQuote("Tesla") }
    try {
        val quote1 = deferred1.await()
    } catch (e: Exception) {
        println("Request failed : $e.")
    }
    val deferred2 = async() { getStockQuote("Aple") }
    try {
        val quote2 = deferred2.await()
    } catch (e: Exception) {
        println("Request failed : $e.")
    }
    val deferred3 = async() { getStockQuote("Google") }
    try {
        val quote3 = deferred3.await()
    } catch (e: Exception) {
        println("Request failed : $e.")
    }
}
```

Exception Handling with supervisorScope

```
/* Because the supervisorScope is used. If one child failed the whole job is NOT cancelled */
val exceptionHandler = CoroutineExceptionHandler { context, exception ->
    println("Exception thrown somewhere within parent or child: $exception.")
}
val job = CoroutineScope(Dispatchers.IO).launch(exceptionHandler) {
 supervisorScope {
    val deferred1 = async() { getStockQuote("Tesla") }
    try {
        val quote1 = deferred1.await()
    } catch (e: Exception) {
        println("Request failed : $e.")
    }
    val deferred2 = async() { getStockQuote("Aple") }
    try {
        val quote2 = deferred2.await()
    } catch (e: Exception) {
        println("Request failed : $e.")
    }
    val deferred3 = async() { getStockQuote("Google") }
    try {
        val quote3 = deferred3.await()
    } catch (e: Exception) {
        println("Request failed : $e.")
    }
 }
}
```

Summary

- Coroutines are suspending functions:
  - computation that can be suspended then resumed
- A coroutine **must** be **started in a Coroutine Scope** to allow **Structured Concurrency** i.e., the coroutine scope:
  - Keeps track of coroutines running in parallel or sequential
  - Has the ability to cancel running coroutines
  - Is notified when a failure happens: the failure of a child coroutine cancels the scope & other children
- Easier **asynchronous** programming
  - Replace callback-based code with sequential code to handle asynchronous long-running tasks without blocking
  - Structure of asynchronous code is the same as synchronous code

Resources

- Kotlin coroutines
  - https://kotlinlang.org/docs/reference/coroutines-overview.html
  - https://developer.android.com/kotlin/coroutines
- Part 1: Coroutines, Part 2: Cancellation in coroutines, and Part 3: Exceptions in coroutines
- Coroutines codelab
  - https://codelabs.developers.google.com/codelabs/kotlin-coroutines